A. van Deursen

Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study

Abstract *The use of a* domain-specific language *can help to develop readable and maintainable applications in that domain with little effort. Alternatively, the same aims can be achieved by setting up an* object-oriented framework. *For the domain of* financial engineering, *independently both an object-oriented framework and a domain-specific language have been developed. We use this opportunity to contrast these two, to highlight the differences and to discuss opportunities for mutual benefits.*

1 Introduction

"If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language" [GHJV94, p. 243]. This idea of introducing a *domain-specific language* (DSL) has been used by many software developers, resulting in languages such as Lex, Yacc, PostScript, AWK, SQL, etc. The benefits of using a DSL include improved readability, maintainability, flexibility, and portability of software [KMB+96, NJ97].

Alternatively, an *object-oriented framework* can be developed to support the construction of families of related programs [JF88, Joh97]. A framework is a set of cooperating (abstract) classes for a given domain a software engineer can use to develop an application in that domain.

Frameworks and DSLs have much in common. In fact, some frameworks are shipped together with a DSL to access the functionality available in the DSL [Joh97]. In this paper, we will try to highlight the differences between the two approaches, in order to arrive at criteria for deciding whether to develop a DSL or a framework and at ways in which DSL and framework development can benefit from each other.

To achieve this, we study a single application domain —financial engineering— for which both an object-oriented framework (the ET++ SwapsManager [EG92, BE93]) and a domain-specific language (Risla [AD92, ADR95, BDK+96, DK97]) have been developed.

Related Work Network protocol software is another area where both a DSL (the *Morpheus* language [AP93]) and a framework (*Conduits+* [HJE95]) have been developed. [HJE95] includes a short comparison, in which it is concluded that "although the framework is unlikely to achieve the same execution efficiency as a special-purpose programming language, it offers similar, but more easily extensible, composition facilities."

2 The Financial Engineering Domain

Financial engineering deals with financial *instruments*, also called *interest rate products*. The aim of these products is to facilitate financial transactions as taking place, e.g., in inter-bank trade or company take-overs. The products can be used to provide protection against interest rate or currency exchange rate fluctuations.

The simplest interest rate product is the loan: a fixed amount in a certain currency is borrowed for a fixed period at a given interest rate. More complicated products, such as the *financial future*, the *forward rate agreement*, or the *capped floater* [Cog95, Chapter 12], all aim at *risk reallocation*. Banks can invent new ways to do this, giving rise to more and more interest rate products. Not surprisingly, different interest rate products have much in common, making financial engineering an area suitable for incorporating domain-specific knowledge in tools, languages, or libraries.

The flexibility of the interest rate product market complicates the task of the software engineer. Software systems dealing with interest rate products include the bank's financial administration (who is buying what), and — more importantly — the management information system allowing decision makers to assess risks involved in the products currently processed. Typical problems found in such systems are that it is too difficult (1) to introduce a new type of product quickly, even if it is very similar to existing ones, and (2) to ensure that the instructions given by the financial engineer are correctly implemented by the software engineer. The first problem leads to a long time-to-market for new products; the second leads to potentially incorrect behavior.

3 The Risla Language

Dutch bank MeesPierson, together with software house CAP Gemini saw the use of a specific language for describing interest rate products as the solution to the problems of long time-to-market and potentially inaccurate implementations. The language was to be readable for financial engineers, and descriptions in this language were to be compiled into COBOL. In this section we summarize earlier (and more detailed) accounts given by [ADR95, BDK+96, DK97] of the development and use of this language.

3.1 Language Development

The development of this language, called RISLA (for Rente Informatie Systeem Language — Interest rate information system language), started in 1992, and can be summarized as follows:

- MeesPierson had a very good library of COBOL routines for operating on cash flows, intervals, interest payment schemes, date manipulations, etc.;
- Using this library directly in COBOL did not provide the right level of abstraction, and cumbersome encoding tricks were needed to use, e.g., lists without a fixed length;

- An interest rate product can be considered as a "class": it contains instance variables to be assigned at creation time (the principal amount, the interest rate, the currency, etc.), information methods for inspecting actual products (when is interest to be paid), and registration methods for recording state changes (pay one redemption).

The language RISLA was designed to describe interest rate products along these lines. An instantiated product is called a *contract*, fixing the actual amount, rate, etc. of a particular product sold. The language is based on a number of built-in data types for representing cash flows, dates, rates, intervals, balances, ..., and has a large number of built-in operations manipulating these data types (the operations correspond to the subroutines in the COBOL library). A product definition specifies the contract parameters, information methods, and registration methods.

RISLA is translated into COBOL. Other systems in the bank can invoke the generated COBOL to create new contracts, to ask information about existing contracts, or to update contract information. The initial version of RISLA was used to define about 30 interest rate products.

After a few years of working with RISLA, the users experienced the modularization features of RISLA as inadequate. A RISLA description defines a complete product; but different products are constructed from similar *components*. To remedy this situation, a project *Modular* RISLA was started. RISLA was extended with a small modular layer, featuring parameterization and renaming. Moreover, a *component library* was developed, and the most important products were described using this library.

In addition to that, the RISLA development team made an effort to make the language more accessible to the financial experts. To that end, an inter-active *questionnaire* interface to the component library was developed. End-users can combine existing components into a new product by filling in the answers of a questionnaire. The answers are used to select the relevant RISLA components. This definition may contain some holes that are specific to this product, which can be filled by writing the appropriate RISLA code. The modular definition is then expanded to a flat (non-modular) definition, which in turn is compiled into COBOL.

As a last point of interest, the actual questionnaire used is defined using a second domain-specific language: RISQUEST. This is a language for defining questions together with permitted answers (choice from a fixed set, free text). Moreover, RISQUEST has constructs for indicating in which order questions are to be asked, and how this sequencing may depend on the actual answers given. Last but not least, RISQUEST can be used to associate library components with the possible answers. A RISQUEST definition is entered in textual form, and it is generated into a Tcl/Tk program. This program can be invoked by a financial engineer to fill in the questionnaire and to generate the corresponding modular RISLA.

3.2 Assessment

The RISLA project has met its targets: the time it costs to introduce a new product is down from an estimated three months to two or three weeks. Moreover, financial engineers themselves can use the questionnaire to compose new products. Furthermore, it has become much easier to validate the correctness of the software realization of the interest rate products. In addition to

that, the component library appears to be useful for other product families, such as insurances or options.

After more than five years after its introduction, the RISLA language is still used actively. RISLA has survived several mergers during its lifetime (at which moment there was a choice between the interest rate systems running at the two merged banks). RISLA could be easily connected to the systems of the new partner by extending the product descriptions with some methods providing the new reports required required by the new systems.

At the negative side, it is not so easy to extend the language. When a new data type or a new built-in function is required, the compiler, as well as the COBOL library, needs to be adapted. This requires skills in compiler construction technology, which is not the typical background of people working mainly in a COBOL environment. Also, the RISLA product definitions have become longer and longer. Whenever there was a new software system requiring information about products that was not provided in the existing methods, new methods had to be added, sometimes requiring new data types or extensions to the RISLA language.

4 The ET++ SwapsManager

The ET++ SwapsManager is a tool for the valuation of Swaps developed at the Union Bank of Switzerland [EG92, BE93]. The challenges addressed are very similar to those of the RISLA project, and include (1) short life cycles (low time-to-market); (2) providing management with accurate "what-if" analyses; (3) taking advantages of similarities between different applications; (4) use of intuitive human/computer interfaces [EG92].

ET++ SwapsManager was a pilot study aimed at showing how various advanced technologies could be useful for building banking applications. Technology to be evaluated included modern user interfaces, object orientation, domain-specific frameworks, and design patterns. This is unlike the RISLA project, the primary purpose of which was to develop a production environment for dealing with interest rate products.

The key abstractions of the ET++ SwapsManager are the financial instrument (the interest rate product), and the *discount function*, which is a way of computing instrument's current value at the financial markets. The instrument is characterized by its cash flows, as in RISLA. The "discount function manager" is used to position a contract in a mix of well-known products with similar cash flows.

The SwapsManager takes advantage of the so-called *strategy pattern* [GHJV94] to abstract from different ways to compute, for instance, the number of days in a year or the current market rate. In RISLA, these strategies are encoded as enumeration types, with values such as "LIBOR" (London Interbank Offered Rate) or "LIBID" (—Bid Rate).

5 Evaluation

Expressiveness A DSL provides a natural way to express the non-technical essence of a particular domain, increasing readability and portability. On the other hand, using a full general-

purpose language as in a framework provides more flexibility in adapting it to specific needs.

Legacy Libraries RISLA is an example of a DSL that was used to access a given existing library. This library was functionally entirely adequate, but written in a legacy language. A DSL can be used to provide access to such legacy libraries. In principle, wrapping could be used to achieve the same effect using an object-oriented language, thus basing a framework on an existing legacy library.

The calling framework A framework often is an active entity: It does not get called (as a library), but it calls functions provided by the application developer. The same situation is easily obtained in a DSL setting: A RISLA product description defines functions for computing cash flows, which are called by other systems running at the bank. The fact that COBOL is generated from RISLA makes it easy for legacy systems to connect to information defined in RISLA.

Overriding Default Behavior So-called *white-box* frameworks allow the application developer to override default behavior using inheritance. Although this could be encoded in a DSL as well, this does not seem as natural as in a framework setting.

Language Technology Interactive DSL development requires tools supporting the rapid prototyping of scanners, parsers, type checkers, interpreters, compilers, etc. A discussion of techniques supporting this is given in [DHK96].

Mutual Benefits When developing a DSL from scratch (rather than developing it to access legacy systems), with freedom of choice for the target language, it is most natural to base the DSL implementation on a domain-specific object-oriented framework. When developing a domain-specific framework, extending it with a DSL to access its functionality has a number of advantages:

- It is a guide to the design of the framework. If there is no way to express a certain class or method as a language construct, it is likely that this class or method does not correspond to a natural concept of the domain.
- It encourages the development of *black-box* frameworks (based on composition) rather than *white-box* frameworks (based on inheritance).
- It gives more abstract access to the framework, hiding (encapsulating) what language is used to implement the framework.

References

[AD92] B.R.T. Arnold and A. van Deursen. Algebraic specification of a language defining interest rate products. CWI, Amsterdam; ORFIS International, Huis ter Heide, 1992.

[ADR95] B. R. T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[AP93] M. B. Abbot and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1), 1993.

[BDK+96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[BE93] A. Birrer and T. Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, volume 707 of *LNCS*, pages 21–35. Springer-Verlag, 1993.

[Cog95] Ph. Coggan. *The Money Machine: How the City Works*. Pinguin, 1995. Third edition.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[DK97] A. van Deursen and P. Klint. Little languages: Little maintenance? In *Proceedings of the first ACM SIGPLAN Workshop on Domain-Specific Languages* [Kam97], pages 109–127.

[EG92] Th. Eggenschwiler and E. Gamma. ET++ SwapsManager: Using object technology in the financial engineering domain. In *OOPSLA'92*, pages 166–177. ACM, 1992. SIGPLAN Notices 27(10).

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[HJE95] H. Hüni, R. Johnson, and R. Engel. A framework for network software. In *OOPSLA'95*, pages 358–369. ACM, 1995. ACM SIGPLAN Notices 30(10).

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Joh97] R. E. Johnson. Components, frameworks, patterns. In M. Harandi, editor, *Proc. of the Symposium on Software Reusability SSR97*, pages 10–17, 1997. ACM SIGSOFT Software Engineering Notes 22(3).

[Kam97] S. Kamin (editor). Proceedings of the first ACM SIGPLAN workshop on Domain-Specific Languages. Computer science report, University of Illinois, 1997.

[KMB+96] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 542–553. IEEE, 1996.

[NJ97] L. Nakatani and M. Jones. Jargons and infocentrism. In *Proceedings of the first ACM SIGPLAN Workshop on Domain-Specific Languages* [Kam97], pages 59–74.

Author's Address

Dr. van Deursen, A.
CWI, P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Tel.: +31 20 592 4075
Fax.: +31 20 592 4199
Email: arie@cwi.nl
URL: `http://www.cwi.nl/~arie/`